آزمایش پنج

خلاصه ی آزمایش:

مروری بر لینوکس و برخی دستورات در آن

اهداف آزمایش:

- آشنایی با برخی دستورات Unix
- آشنایی با shell script
- آشنایی با Regex
- آشنایی با AWK
- آشنایی با Sed

تجهیزات مورد نیاز:

- برد Raspberry pi II و کابل برق آن
- کارت Micro SD
- کامپیوتر (به همراه موس و کیبورد)
- کابل شبکه جهت ارتباط برد با کامپیوتر
- نرم افزار Putty جهت راه اندازی SSH

شرح آزمایش:

1- بخش " آشنایی با برخی دستورات Unix" را مطالعه کنید.

2- بخش " آشنایی با shell script" را مطالعه کنید و در صورت لزوم، کد هایی را که در این بخش نشان داده شده اند را تست کنید.

3- اسکریپتی بنویسید که عدد n را دریافت کند و عدد متناظر آن در سری فیبوناچی را نشان دهد.

4- اسکریپتی بنویسید که سری لغات را از از user بخواهد و لغات آن را با متد insertion sort در خروجی به صورت مرتب نمایش دهد.

```
  GNU nano 2.2.6                File: strcmp.sh

#!/bin/bash

str1="hello1"
str2="hello"
if [[ "$str1">="$str2" ]]
then
        echo "gt"
else
        echo "lt"
fi
```

```
  GNU nano 2.2.6                File: readArr.sh

#!/bin/bash

read -a ARRAY
printf '%s\n' "${ARRAY[@]}"
```

5- بخش " آشنایی با Regex" را مطالعه کنید.

6- فایل زیر را به نحوی تغییر دهید که اینکه داده ی ورودی، آدرس IP است یا خیر را تشخیص دهد.

```
  GNU nano 2.2.6               File: IsIPAddress.sh

#!/bin/bash

if [ $# != 1 ]; then
    echo "Usage: $0 address"
    exit 1
else
    ip=$1
fi

if [[ $ip =~ ****************************** ]]; then
    echo "Looks like an IPv4 IP address"
else
    echo "oops"
fi
```

7- بخش " آشنایی با AWK" را مطالعه کنید.

8- فایل awk01 را به نحوی اصلاح کنید که با اجرای دستور زیر، آدرس IP برای eth0 در دستگاه را به همراه net mask نمایش داده شود

```
pi@raspberrypi ~/session05/awk $ cat /etc/network/interfaces | awk -f awk01
```

```
  GNU nano 2.2.6                        File: awk01

BEGIN{
print "start"
}
{
print $0
}
END{}
```

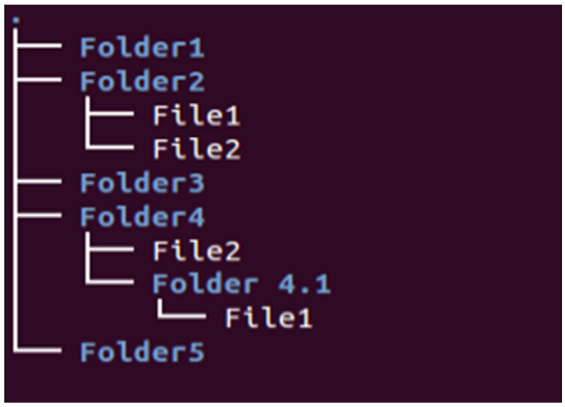
9- یک اسکریپت بنویسید که:

1- یک فولدر را به عنوان ورودی دریافت کند

2- چک کند که حتما ورودی به اسکریپت داده شده باشد

3- چک کند که ورودی معتبر باشد

4- تمام فولدر های زیر مجموعه ی داده ی ورودی را مانند شکل زیر به شکل درختی نمایش دهد.



برای بدست آوردن لیست از فولدر ها می توانید از دستور زیر استفاده کنید.

```
 GNU nano 2.2.6                File: tree.sh

#!/bin/bash
cd $1
ls -R | grep ":$"
```

وظایف:

1- بخش مطالعه ی این دستور کار به طور کامل مطالعه شود.

مطالعه:

آشنایی با برخی دستورات Unix

آشنایی با shell script

آشنایی با Regex

آشنایی با AWK

آشنایی با Sed

آشنایی با برخی دستورات Unix

Starting with Unix

- When you enter your password, the letters that you type are not displayed on your terminal for security reasons.
- UNIX is case sensitive, so make sure that the case of letters is matched exactly to those of your password.
- Depending on how your system is set up, you should then see either a $, a % or another prompt. That is your default shell prompting you to provide some course of action.
- Here's an example login:

```
UNIX®  System V  Release 4.0
login: glass
Password:              --> What is typed here is secret and doesn't show.
Last login: Sun  Feb  15  18:33:26  from dialin
$ _
```

Command	Description
halt	Brings the system down immediately.
init 0	Powers off the system using predefined scripts to synchronize and clean up the system prior to shutdown
init 6	Reboots the system by shutting it down completely and then bringing it completely back up
poweroff	Shuts down the system by powering off.
reboot	Reboots the system.
shutdown	Shuts down the system.

Logging out

- To leave the UNIX system, press the keyboard sequence Control-D at your shell prompt.
- This command tells your login shell that there is no more input for it to process, causing it to disconnect you from the UNIX system.
- Most systems then display a "login:" prompt and wait for another user to log in.

```
$ ^D
UNIX® System V Release 4.0
login:                          --> wait for another user to log in.
```

- If you connect to a remote server through ssh connection, you will not see the new login prompt; instead, you will be disconnected.

Shells

- The $ or % prompt that you see when you first log in is displayed by a special kind of program called a shell.
- A Shell is a program that acts as a middleman between you and the raw UNIX operating system.
- It lets you run programs, build pipelines of processes, save output to files, and run more that one program at the same time.
- A shell executes all of the commands that you enter.
- The four most popular shells are:
  - the Bourne shell (sh)
  - the Korn shell (ksh)
  - the C shell (csh)
  - the Bash Shell (bash)

date [yymmddhhmm[.ss]]

- Without any arguments, date displays the current date and times.
- If arguments are provided, date sets the date to the supplied setting, where:
  - yy is the last two digits of the year,
  - the first mm is the number of the month,
  - dd is the number of the day,
  - hh is the number of hours (using the 24-hour clock), and
  - the last mm is the number of minutes.
- The optional ss is the number of seconds. Only a super-user may set the date.

clear

- This utility clears your screen.

Using special characters

- Some characters are interpreted specially when typed at a UNIX terminal.
- These characters are sometimes called metacharacters and may be listed by using the stty utility with the -a (all)option.
- Here's an example of the use of the stty utility for listing metacharacters:

```
$ stty  -a                          --> obtain terminal line settings
speed 38400 baud; rows 50; columns 80; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D; eol = <undef>;
eol2 = <undef>; start = ^Q; stop = ^S; susp = ^Z; rprnt = ^R; werase = ^W;
lnext = ^V; flush = ^O; min = 1; time = 0;
-parenb -parodd cs8 -hupcl -cstopb cread -clocal -crtscts
-ignbrk -brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl ixon -ixoff
-iuclc -ixany  -imaxbel
opost -olcuc -ocrnl onlcr -onocr -onlret -ofill -ofdel nl0 cr0 tab0 bs0 vt0 ff0
isig icanon iexten echo echoe echok -echonl -noflsh -xcase -tostop -echoprt
echoctl echoke
```

Meta-characters

- The carat(^) in front of each letter means that the Control key must be pressed at the same time as the letter.
- The default meaning of each option is as follows:

Meta-character	Meaning
erase	Backspace one character
kill	Erase all of the current line.
werase	Erase the last word.
rprnt	Reprint the line.
flush	Ignore any pending input and reprint the line.
lnext	Don't treat the next character specially.
susp	Suspend the process for a future awakening.
intr	Terminate (interruput) the foreground job with no core dump.
quit	Terminate the foreground job and generate a core dump.
stop	Stop/restart terminal output.
eof	End of input.

Terminating a Process: Control-c

- There are often times when you run a program and then wish to stop it before it's finished.
- The standard way to execute this action in UNIX is to press the keyboard sequence Control-C.
- Most processes are immediately killed and your shell prompt is returned.
- Here's an example of the use of Control-C:

```
$ man chmod
 CHMOD(1V)   USER   COMMANDS   CHMOD(1V)
 NAME
       chmod - change the permissions mode of a file
^C
$ _
```

Pausing Output to Terminal: Control-s/Control-q

- If the output of a process starts to rapidly scroll up the screen, you may pause it by pressing Control-S.
- To resume the output, you may either press Control-s again or press Control-q.
- This sequence of control characters is sometimes called XON/XOFF protocol.
- Here's an example of its use:

```
$ man chmod
 ...
 ^s                    ---> suspend terminal output.
 ^q                    ---> resume terminal output.
 ...
$ _
```

End of Input: Control-d

- You must tell the utility when the input from the keyboard is finished.
- To do so, press Control-D on a line of its own after the last line of input. Control-D signifies the end of input.

- For example, the mail utility allows you to send mail from the keyboard to a named user:

```
$ mail tim                    --> send mail to my friend tim.
Hi Tim,                       --> input is entered from the keyboard.
I hope you get this piece of mail. How about building a country
one of these days?

- with best wishes from Graham
^D                            --> tell the terminal that there's no more input.
$ _
```

Setting/Changing Password: passwd

- passwd allows you to change your password.
- You are prompted for your old password and then twice for the
- new one.
- The new password may be stored in an encrypted form in the
- password file "/etc/passwd" or in a "shadow" file (for more
- security), depending on your version of UNIX.

- An example of changing a password:

```
$ passwd
  Current password : penguin
  New password( ? For help ) : GWK145W          --> invisible
  New password( again ) : GWK145W               --> invisible
  Password changed for glass
  $ _
```

- Note that you wouldn't normally be able to see the passwords, as UNIX turns off the keyboard echo when you enter them.

working with directories

Printing Working Directory: pwd

- To display your shell's current working directory, use the pwd utility, which works like this:

```
UNIX®  System V Release 4.0

login : glass
Password : ......

$pwd
/home/glass
$ _
```

Changing Directories: cd

- cd [directoryName]

- The following might be inconvenient; especially if we deal with large hierarchy:

```
$ vi   lyrics/heart.ver1                --> invoke the vi editor
```

- Instead, change directory:

```
$ cd  lyrics                            --> change directory
$ vi   heart.ver1                       --> invoke the vi editor
```

- The cd shell command changes a shell's current working directory to be directoryName.

- If the directoryName argument is omitted, the shell is moved to its owner's home directory.

Listing Contents of a Directory: ls

- The ls utility, which lists information about a file or a directory.

ls -adgIsFR { fileName }* {directoryName}*

- ls lists all of the files in the current working directory in alphabetical order, excluding files whose names start with a period.
- The -a option causes such files to be included in the listing.
- The -d option causes the details of the directories themselves to be listed, rather than their contents.
- The -g option list a file's group.
- The -l option generates a long listing, including permission flags, the file's owner, and the last modification time.

Listing Contents of a Directory

- The -s option causes the number of disk blocks that the file occupies to be included in the listing. (A block is typically between 512 and 4K bytes.)
- The -F option causes a character to be placed after the file's name to indicate the type of the file:
  - * means an executable file, / means a directory file,
  - @ means a symbolic link, and = means a socket.
- The -R option recursively lists the contents of a directory and its subdirectories.

Directory Listing

- Here's an example of the use of ls :

```
$ ls                          --> list all files in current directory.
heart

$ ls -l heart                          --> long listing of "heart."
-rw-r--r--   1   glass   106   Jan 30 19:46   heart
$ _
```

the name of the file
the time that the file was last modified
the size of the file, in bytes
the username of the owner of the file
the hard-link count
permission mode of the file

Listing Contents of a Directory

- You may obtain even more information by using additional options:

```
$ ls -algFs                          --> extra-long  listing of current dir
total 3                              --> total number of blocks of storage.
1 drwxr-xr-x     3     glass   cs   512   Jan  30  22:52 ./
1 drwxr-xr-x    12     root    cs   1024  Jan  30  19:45 ../
1 -rw-r--r--     1     glass   cs   106   Jan  30  19:46 heart
$ _
```

- The -s option generates an extra first field, which tells you how many disk blocks the file occupies.
- On some UNIX systems, each disk block is 1024 bytes long, which implies that a 106-byte file actually takes up 1024 bytes of physical storage.

Making Directory: mkdir

mkdir -p newDirectoryName

- The mkdir utility creates a directory. The -p option creates any parent directories in the newDirectoryName pathname that do not already exist.
- If newDirectoryName already exists, an error message is displayed and the existing file is not altered in any way.

```
$ mkdir lyrics          --> creates a directory called "lyrics".
$ ls –lF                --> check the directory listing in order
                        --> to confirm the existence of the
                        --> new directory.
-rw-r--r--   1  glass  106  Jan 30 23:28  heart.ver1
drwxr-xr-x   2  glass  512  Jan 30 19:49  lyrics/
$ _
```

Deleting a Directory: rmdir

rmdir { directoryName }+

- The rmdir utility removes all of the directories in the list of directory names provided in the command. A directory must be empty before it can be removed.
- To recursively remove a directory and all of its contents, use the rm utility with the -r option (see next slide).
- Here, we try to remove the "lyrics.draft" directory while it still contains the draft versions, so we receive the following error message:

```
$ rmdir lyrics.draft
rmdir: lyrics.draft: Directory not empty.
$ _
```

Deleting Directories: rm -r

- The rm utility allows you to remove a file's label from the hierarchy.
- Here's a description of rm:

rm -fir {fileName}*

- The rm utility removes a file's label from the directory hierarchy.
- If the filename doesn't exist, an error message is displayed.
- The -i option prompts the user for confirmation before deleting a filename. It is a very good idea to use this option or create a shell alias that translates from "rm" to "rm –i". If you don't, you will loose some files one day – you have been warned!
- If fileName is a directory, the -r option causes all of its contents, including subdirectories, to be recursively deleted.
- The -f option inhibits all error messages and prompts. It overrides the –i option (also one coming from an alias). This is dangerous!

Removing Directories with Files

- The -r option of rm can be used to delete the "lyrics.draft" directory and all of its contents with just one command:

```
$ cd                    --> move to my home directory.
$ rm -r  lyrics.draft   --> recursively delete directory.
$ _
```

working with files

Renaming/Moving a File: mv

```
mv  -i  oldFileName  newFileName
mv  -i  {fileName}*  directoryName
mv  -i  oldDirectoryName newDirectoryName
```

- The first form of mv renames oldFileName as newFileName.
- The second form allows you to move a collection of files to a directory.
- The third form allows you to move an entire directory.
- The -i option prompts you for confirmation if newFileName already exists so that you do not accidentally replace its contents. You should learn to use this option (or set a convenient shell alias that replaces "mv" with "mv –i"; we will come back to this later).

Copying Files: cp

- To copy the file, I used the cp utility, which works as follows:

```
cp -i oldFileName  newFileName
cp -ir { fileName }*  directoryName
```

- The first form of cp copies the contents of oldFileName to newFileName.
- If the label newFileName already exists, its contents are replaced by the contents of oldFileName.
- The -i option prompts you for confirmation if newFileName already exists so that you do not accidentally overwrite its contents. Like with mv, it is a good idea to use this option or create an alias.

Copying Files: cp

- The -r option causes any source files that are directories to be recursively copied, thus copying the entire directory structure.

- cp actually does two things
  - It makes a physical copy of the original file's contents.
  - It creates a new label in the directory hierarchy that points to the copied file.

```
$ cp heart.ver1  heart.ver2 --> copy to "heart.ver2".
$ ls -l heart.ver1  heart.ver2          --> confirm the existence of both files.
-rw-r--r--  1  glass  106  Jan 30 23:28  heart.ver1
-rw-r--r--  1  glass  106  Jan 31    00:12  heart.ver2
$ cp -i heart.ver1  heart.ver2          --> what happens?
```

Creating a file with cat

cat -n {fileName}*

- The cat utility takes its input from standard input or from a list of files and displays them to standard output.

- The -n option adds line numbers to the output. cat is short for "concatenate" which means "to connect in a series of links."

- By default, the standard input of a process is from the keyboard and the standard output is to the screen.

```
$ cat  > heart      --> store keyboard input into a file called "heart".
I hear her breathing,
I'm  surrounded by the sound.
Floating in this secret place,
I never shall be found.
^D                  --> tell cat that the end of input has been reached.
$ _
```

Displaying a File: cat

- cat with the name of the file that you wanted to display:

```
$ cat heart                    --> list the contents of the "heart" file.
I hear her breathing.
I'm surrounded by the sound.
Floating in this secret place,
I never shall be found.
$ _
```

- cat is good for listing the contents of small files, but it doesn't pause between full screens of output.

Displaying a File: more

more -f [+lineNumber] { fileName }*

- The more utility allows you to scroll a list of files, one page at a time.
- By default, each file is displayed starting at line 1, although the +option may be used to specify the starting line number.
- The -f option tells more not to fold (or wrap) long lines.

- After each page is displayed, more displays the message "--more--" to indicate that it's waiting for a command.

- To list the next page, press the space bar.
- To list the next line, press the Enter key.
- To quit from more, press the "q" key.
- ^B will display the previous page
- H will display help page

- Try:

```
$ ls –la /usr/bin > myLongFile
$ more myLongFile
```

Displaying a File: head and tail

head -n { fileName }*

- The head utility displays the first n lines of a file. If n is not specified, it defaults to 10. If more than one file is specified, a small header identifying each file is displayed before its contents.

tail -n { fileName }*

- The tail utility displays the last n lines of a file. If n is not specified, it defaults to 10. If more than one file is specified, a small header identifying each file is displayed before its contents.

- The first two lines and last two lines of my "heart" file.

```
$ head -2 heart                    --> list the first two lines.
I hear her breathing,
I'm surrounded by the sound.
$ tail -2 heart                    --> list the last two lines.
Floating in this secret place,
I never shall be found.
$ head -15 myLongFile              --> see what happens
```

Counting Lines, Words and Characters in Files: wc

wc -lwc {fileName}*

- The wc utility counts the number of lines, words, and/or characters in a list of files.
- If no files are specified, standard input is used instead.

- The -l option requests a line count,
- the -w option requests a word count,
- and the -c option requests a character count.

- If no options are specified, then all three counts are displayed.
-
- A word is defined by a sequence of characters surrounded by tabs, spaces, or new lines.

File attributes

File Attributes

- File Storage
  - The number of blocks of physical storage taken up by the file is shown in field 1 and is useful if you want to know how much actual disk space a file is using.
- Filenames
  - The name of the file is shown in field 8. A UNIX filename may be up to 255 characters in length.
  - The only filenames that you definitely can't choose are "." and "..", as these names are predefined filenames that correspond to your current working directory and its parent directory, respectively. One can use " " (blank, space) in file names, but it is difficult to deal with files like that. Using special characters will definitely confuse your shell at some point.
- Time of Last File Modification
  - the time that the file was last modified and is used by several utilities.

File Attributes

- File Owner
  - Field 3 tells you the owner of the file. Every UNIX process has an owner, which is typically the same as the username of the person who started it.
  - the string of text known as the username is typically how we refer to a user, UNIX represents this identity internally as an integer known as the user ID.
  - The username is easier for humans to understand than a numeric ID.
- File Group
  - Field 5 shows the file's group. Every UNIX user is a member of a group. This membership is initially assigned by the system administrator and is used as part of the UNIX security mechanism.

File Attributes

- File Types
  - Field 2 describes the file's type and permission settings.

- In ls –lgsF example:

- 1 -rw-r--r-- 1 glass cs 213 Jan 31 00:12 heart.final

- The first character of field 2 indicates the type of file, which is encoded as follows:

character	File Type
-	regular file
d	directory file
b	buffered special file(such as a disk drive)
c	unbuffered special file(such as a terminal)
l	symbolic link
p	pipe
s	socket

Determining Type of a File: file

file { fileName }+

- The file utility attempts to describe the contents of the fileName argument(s), including the language in which any of the text is written.

- file is not reliable; it may get confused.

- When file is used on a symbolic-link file, file reports on the file that the link is pointing to, rather than on, the link itself.

- For example,

```
$ file heart.final                    --> determine the file type.
heart.final: ascii text
$ _
```

File Permissions

File Permissions (Security)

- File permissions are the basis for file security. They are given in three clusters. In the example, the permission settings are "rw-r--r--":

- 1 -rw-r--r-- 1 glass cs 213 Jan 31 00:12 heart.final

User (owner)	Group	Others
rw-	r--	r-- ← clusters

Each cluster of three letters has the same format:

Read permission	Write permission	Execute permission
r	w	x

File Permission

- The meaning of the read, write, and execute permissions depends on the type of file:

	Regular file	Directory file	Special file
Read	read the contents	read the directory (list the names of files that it contains)	read from the file using the read() system call.
Write	change the contents	Add or remove files to/from the directory	write to the file using the write() system calls.
Execute	execute the file if the file is a program	access files in the directory	No meaning.

Change File Permissions: chmod

chmod -R change{, change}* {fileName}+

- The chmod utility changes the modes (permissions) of the specified files according to the change parameters, which may take the following forms:

clusterSelection+newPermissions (add permissons)
clusterSelection-newPermissions (subtract permissions)
clusterSelection=newPermissions (assign permissions absolutely)

- where clusterSelection is any combination of:
  - u (user/owner)
  - g (group)
  - o (others)
  - a (all)

  and newPermissions is any combination of
  - r (read)
  - w (write)
  - x (execute)
  - s (set user ID/set group ID)

Changing File Permissions

- The -R option recursively changes the modes of the files in directories.
- Note that changing a directory's permission settings doesn't change the settings of the files that it contains.
- To remove read permission from others, we used chmod as follows:

```
$ ls -lg heart.final          --> to view the settings before the change.
-rw-r-----  1  glass  music 213 Jan 31 00:12 heart.final
$ chmod g-r heart.final
$ ls -lg heart.final
-rw-------  1  glass  music 213 Jan 31 00:12 heart.final
$ _
```

Changing File Permissions: examples

Requirement	Change parameters
Add group write permission	g+w
Remove user read and write permission	u-rw
Add execute permission for user, group, and others.	a+x
Give the group read permission only.	g=r
Add writer permission for user, and remove group read permission.	u+w,g-r

Changing File Permissions Using Octal Numbers

- The chmod utility allows you to specify the new permission setting of a file as an octal number.
- Each octal digit represents a permission triplet.

For example, if you wanted a file to have the permission settings of

rwxr-x---

then the octal permission setting would be 750, calculated as follows:

	User	Group	Others
setting	rwx	r-x	---
binary	111	101	000
octal	7	5	0

Changing File Owner: chown

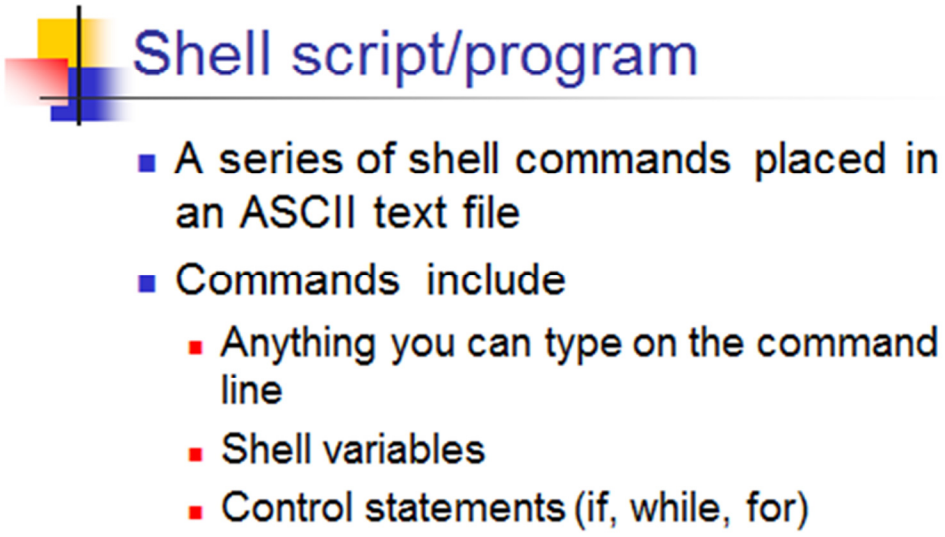
chown -R newUserId {fileName}+

- The chown utility allows a super-user to change the ownership of files (some Unix versions allow the owner of the file to reassign ownership to another user). All of the files that follow the newUserId argument are affected.
- The -R option recursively changes the owner of the files in directories.
- Example: change the ownership of "heart.final" to "tim" and then back to "glass" again:

```
$ ls -lg heart.final                    --> to view the owner before the change.
-rw-r-----  1 glass  music 213 Jan 31  00:12  heart.final
$ chown  tim heart.final                --> change the owner to "tim".
$ ls -lg heart.final              --> to view the ownership after the change.
-rw-r-----  1 tim    music 213 Jan 31  00:12  heart.final
$ chown glass heart.final --> change the owner back to "glass".
$ _
```

آشنایی با shell script

Shell script چیست؟



تفاوت برنامه نویسی با اسکریپت نویسی:

Programming or Scripting ?

- bash is not only an excellent command line shell, but a scripting language in itself. Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.

- Difference between programming and scripting languages:

  – Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.
  – A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction. Interpreted programs are generally slower than compiled programs. The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Other examples of scripting languages are Perl, Lisp, and Tcl.

نحوه ی اجرای یک اسکریپت:

Script execution

- Provide script as an argument to the shell program (e.g. `bash my_script`)
- Or specify which shell to use within the script
  - First line of script is `#!/bin/bash`
  - Make the script executable using `chmod`
  - Make sure the PATH includes the current directory
  - Run directly from the command line
- No compilation is necessary!

```
  GNU nano 2.2.6                File: hello.sh

#!/bin/bash

echo "hello world!"
cd ~
pwd
```

```
pi@raspberrypi ~/session05/bash $ bash hello.sh
hello world!
/home/pi
pi@raspberrypi ~/session05/bash $
```

نحوه ی دیباگ کردن:

Debugging on the entire script

When things don't go according to plan, you need to determine what exactly causes the script to fail. Bash provides extensive debugging features. The most common is to start up the subshell with the `-x` option, which will run the entire script in debug mode. Traces of each command plus its arguments are printed to standard output after the commands have been expanded but before they are executed.

```
pi@raspberrypi ~/session05/bash $ bash -x hello.sh
+ echo 'hello world!'
hello world!
+ cd /home/pi
+ pwd
/home/pi
pi@raspberrypi ~/session05/bash $
```

Debugging on part(s) of the script

Using the **set** Bash built-in you can run in normal mode those portions of the script of which you are sure they are without fault, and display debugging information only for troublesome zones.

Overview of set debugging options

Short notation	Long notation	Result
set -f	set -o noglob	Disable file name generation using metacharacters (globbing).
set -v	set -o verbose	Prints shell input lines as they are read.
set -x	set -o xtrace	Print command traces before executing command.

The dash is used to activate a shell option and a plus to deactivate it. Don't let this confuse you!

```
  GNU nano 2.2.6                File: hello_debug.sh

#!/bin/bash

echo "hello world!"
echo

set -x
echo "debug is activated"
cd ~
set +x

echo
pwd
```

```
pi@raspberrypi ~/session05/bash $ bash hello_debug.sh
hello world!

+ echo 'debug is activated'
debug is activated
+ cd /home/pi
+ set +x

/home/pi
pi@raspberrypi ~/session05/bash $
```

Quoting

- Quoting is necessary to use special characters in a variable's value or string
- "" - shell only interprets $ and `` and \
  - $ - variable substitution
  - ` - Command substitution
  - \" - Literal double quote
- \ is used to escape characters
  - echo ``date +%D`` will print: 10/06/03
- '' - shell doesn't interpret special characters
  - echo ``date +%D`` will print: `date +%D`

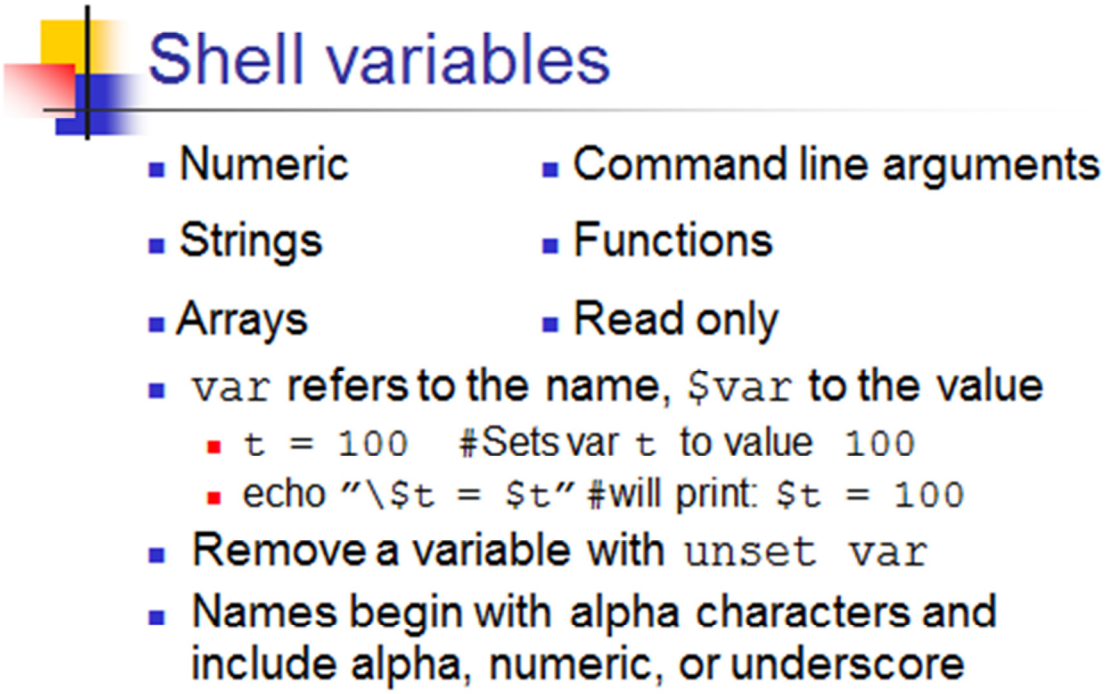
```
pi@raspberrypi ~/session05 $ echo `date +%D`
11/02/15
pi@raspberrypi ~/session05 $ echo 'date +%D'
date +%D
pi@raspberrypi ~/session05 $ _
```

```
  GNU nano 2.2.6                File: test01.sh

#!/bin/bash

MYVAR=sometext
echo "double quotes gives you $MYVAR"
echo 'single quotes gives you $MYVAR'
```

متغیر ها :



Using the declare built-in

Using a **declare** statement, we can limit the value assignment to variables.

The syntax for **declare** is the following:

declare OPTION(s) VARIABLE=value

`-a`	Variable is an array.
`-f`	Use function names only.
`-i`	The variable is to be treated as an integer; arithmetic evaluation is performed when the variable is assigned a value (see Section 3.4.6).
`-p`	Display the attributes and values of each variable. When `-p` is used, additional options are ignored.
`-r`	Make variables read-only. These variables cannot then be assigned values by subsequent assignment statements, nor can they be unset.
`-t`	Give each variable the *trace* attribute.
`-x`	Mark each variable for export to subsequent commands via the environment.

Using + instead of − turns off the attribute instead. When used in a function, **declare** creates local variables.

Warning !

- The shell programming language does not type-cast its variables. This means that a variable can hold number data or character data.

```
count=0
count=Sunday
```

- Switching the TYPE of a variable can lead to confusion for the writer of the script or someone trying to modify it, so it is recommended to use a variable for only a single TYPE of data in a script.

- \ is the bash escape character and it preserves the literal value of the next character that follows.

```
$ ls \*
ls: *: No such file or directory
```

```
  GNU nano 2.2.6                File: helloVar.sh

#!/bin/bash

STR="Hello World!"
echo "\$STR = $STR"
```

Numeric variables

- Integer variables are the only pure numeric variables that can be used in bash
- Declaration and setting value:

```
declare -i var=100
```

- Expressions in the style of C:
  - `(( expression ))`
  - `e.g. (( var+=1 ))`
- +, -, *, /, %, &, |, ~, <, >, <=, >=, ==, !=, &&, ||

String variables

- If you do not use the `declare` keyword with option `-i` when using a variable for the first time, it will be a string
- `var=100` makes `var` the string '100'.
  - However, `(( var=100 ))` will treat var as an integer even though it is a string

Array variables

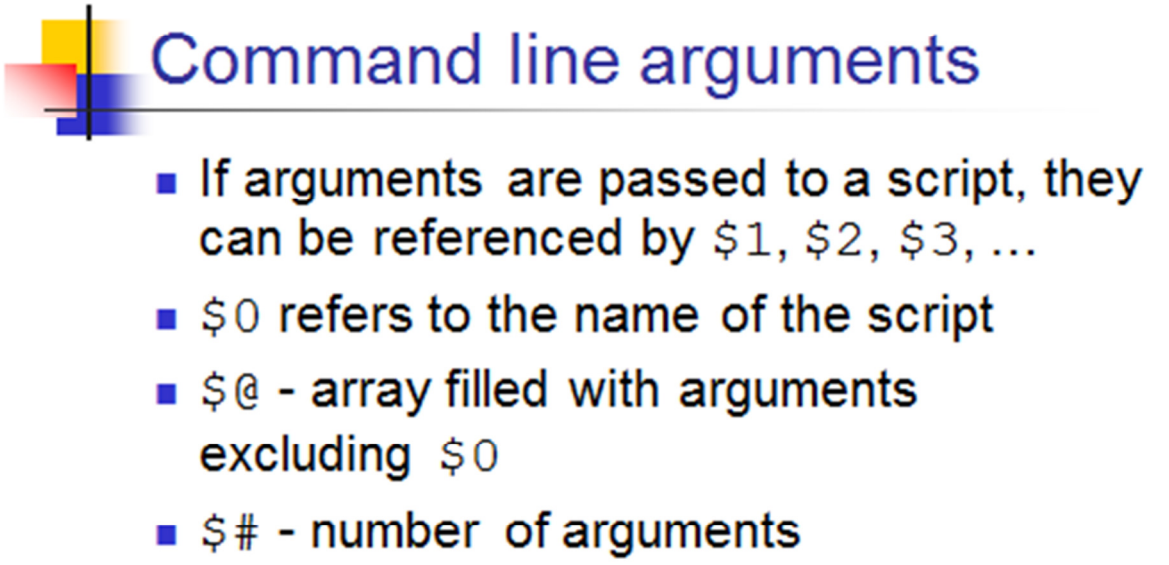
- Array is a list of values
  - Don't have to declare size
- Reference a value by `${name[index]}`
  - `${a[3]}`
  - `$a` (same as `${a[0]}`)
- Use the `declare -a` command to declare an array
  - `declare -a sports`
  - `sports=(ball frisbee puck)`
  - `sports[3]=bat`

Arrays

- Array initialization
  - `sports=(football basketball)`
  - `moresports=(${sports[*]} tennis)`
- `${array[@]}` or `${array[*]}` refers to the entire array contents
- echo `${moresports[*]}` produces

```
football basketball tennis
```

ورودی ها به اسکریپت:



A shift statement is typically used when the number of arguments to a command is not known in advance, for instance when users can give as many arguments as they like. In such cases, the arguments are usually processed in a **while** loop with a test condition of ((**$#**)). This condition is true as long as the number of arguments is greater than zero. The `$1` variable and the **shift** statement process each argument. The number of arguments is reduced each time **shift** is executed and eventually becomes zero, upon which the **while** loop exits.

```
  GNU nano 2.2.6                File: shift.sh

#!/bin/bash

USAGE="Usage: $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
        echo "$USAGE"
exit 1
fi

while (( "$#" )); do
if [[ $(ls "$1") == "" ]]; then
        echo "Empty directory, nothing to be done."
else
        echo "$1 is not empty"

fi

shift

done
```

Exporting variables

- The `export` command, when used with a variable name, allows child processes of the shell to access the variable

```
pi@raspberrypi ~ $ x=hello
pi@raspberrypi ~ $ bash
pi@raspberrypi ~ $ echo $x

pi@raspberrypi ~ $ exit
exit
pi@raspberrypi ~ $ export x
pi@raspberrypi ~ $ bash
pi@raspberrypi ~ $ echo $x
hello
pi@raspberrypi ~ $ x=ciao
pi@raspberrypi ~ $ exit
exit
pi@raspberrypi ~ $ echo $x
hello
pi@raspberrypi ~ $
```

Output

- We have already seen `echo`
- More common in other shells including ksh is `print` (does not exist in bash)
- `echo -n` does not print newline after output

Return values

- Scripts can return an integer value
- Use `exit N`
- The variable `$?` will contain the return value of the last command run
- Can be used to test conditions

Conditions

- If using integers: `(( condition ))`
- If using strings: `[[ condition ]]`
- Examples:
  - `(( a == 10 ))`
  - `(( b >= 3 ))`
  - `[[ $1 = -n ]]`
  - `[[ ($v != fun) && ($v != games) ]]`
- Special conditions for file existence, file permissions, ownership, file type, etc.

Conditions (continued…)

- `[[ -e $file]]` – File exists?
- `[[ -f $file]]` – Regular file?
- `[[ -d $file]]` – Directory?
- `[[ -L $file]]` – Symbolic link?
- `[[ -r $file]]` – File has read permission?
- `[[ -w $file]]` – File has write permission?
- `[[ -x $file]]` – File has execute perm?
- `[[ -p $file]]` – File is a pipe?

If statements

- Syntax:

```
if condition
then
  statements
elif condition
then
  statements
else
  statements
fi
```

(elif … then … statements and else … statements: optional)

If statement

- Example

```
if [[ -r $fname ]]
then
  echo '$fname is readable'
elif [[ -w $fname && -x $fname ]]
then
  echo '$fname is writeable and
  executable'
fi
```

```
  GNU nano 2.2.6                File: if.sh

#!/bin/bash
fname="hello.sh"

if [[ -r $fname ]]
then
        echo '$fname is readable'
elif [[ -w $fname && -x $fname ]]
then
        echo '$fname is writable and executable'
fi
```

For loops

- Syntax:

```
for var [in list]
do
  statements
done
```

- If *list* is omitted, `$@` is assumed
- Otherwise `${list[*]}`
  - where *list* is an array variable.

```
 GNU nano 2.2.6                File: loop.sh

#!/bin/bash

for colors in Red Blue Green Yellow Orange Black Gray White
do
        echo $colors
done
echo
```

```
 GNU nano 2.2.6                File: loop02.sh

#!/bin/bash

for n in {a..d}
do
    echo $n
done
```

While loops

- Syntax:

```
while condition
do
  statements
done
```

- The keywords `break`, `continue`, and `return` have the same meaning as in C/C++

Case statements

- Syntax:

```
case expression in
  pattern1)
    statements ;;
  pattern2)
    statements ;;
  …
  *)
    statements ;;
esac
```

```
  GNU nano 2.2.6                File: case.sh

#!/bin/bash

case $1 in
-a)
        echo "statements related to option a"
        ;;
-b)
        echo "statements related to option b"
        ;;
*)
        echo "all other options"
        ;;
esac
```

Command substitution

- Use the output of a command in a variable, condition, etc. by:
  - `command`
  - $(command)

تمرین: اسکریپتی بنویسید که عدد n را دریافت کند و عدد متناظر آن در سری فیبوناچی را نشان دهد.

Common Shell Features

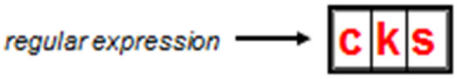
Command	Meaning
>	Redirect output
>>	Append to file
<	Redirect input
<<	"Here" document (redirect input)
|	Pipe output
&	Run process in background.
;	Separate commands on same line
*	Match any character(s) in filename
?	Match single character in filename
[ ]	Match any characters enclosed
( )	Execute in subshell
` `	Substitute output of enclosed command
" "	Partial quote (allows variable and command expansion)
' '	Full quote (no expansion)
\	Quote following character
$var	Use value for variable
$$	Process id

$0	Command name
$n	nth argument (n from 0 to 9)
#	Begin comment
bg	Background execution
break	Break from loop statements
cd	Change directories
continue	Resume a program loop
echo	Display output
eval	Evaluate arguments
exec	Execute a new shell
fg	Foreground execution

jobs	Show active jobs
kill	Terminate running jobs
newgrp	Change to a new group
shift	Shift positional parameters
stop	Suspend a background job
suspend	Suspend a foreground job
time	Time a command
umask	Set or list file permissions
unset	Erase variable or function definitions
wait	Wait for a background job to finish

آشنایی با Regex

Regular Expressions: Exact Matches



UNIX Tools rocks.

match

UNIX Tools sucks.

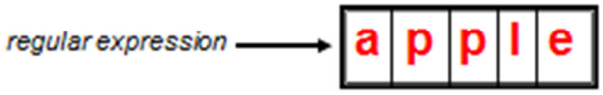
match

UNIX Tools is okay.

no match

Regular Expressions: Multiple Matches

- A regular expression can match a string in more than one place.

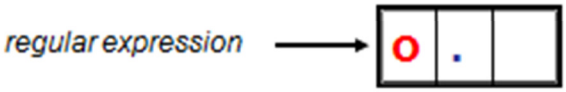


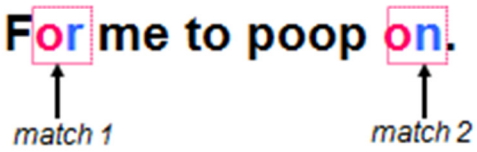
Scrapple from the apple.

match 1 match 2

Regular Expressions: Matching Any Character

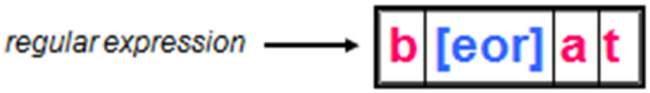
- The . regular expression can be used to match any character.





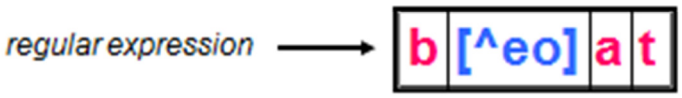
Regular Expressions: Alternate Character Classes

- Character classes [] can be used to match any specific set of characters.



Regular Expressions: Negated Character Classes

- Character classes can be negated with the `[^]` syntax.

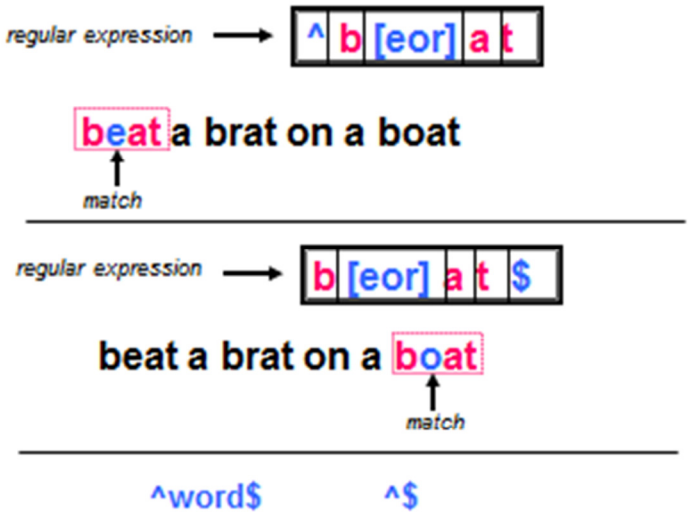


Regular Expressions: Other Character Classes

- Other examples of character classes:
  - [aeiou] will match any of the characters a, e, i, o, or u
  - [kK]orn will match korn or Korn
- Ranges can also be specified in character classes
  - [1-9] is the same as [123456789]
  - [abcde] is equivalent to [a-e]
- You can also combine multiple ranges
  - [abcde123456789] is equivalent to [a-e1-9]
- Note that the - character has a special meaning in a character class but only if it is used within a range
  - [-123] would match the characters -, 1, 2, or 3

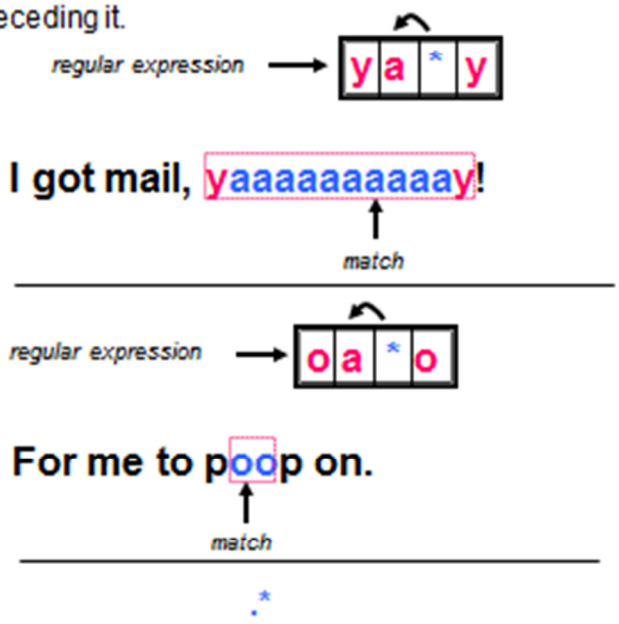
Regular Expressions: Anchors

- Anchors are used to match at the beginning or end of a line (or both).
- ^ means beginning of the line
- $ means end of the line



Regular Expression: Repetions

- The * is used to define **zero or more** occurrences of the *single* regular expression preceding it.



Introduction to Regular Expressions

UNIX

Regular Expressions: Repetion Ranges, Subexpressions

- Ranges can also be specified
  - `{n,m}` notation can specify a range of repetitions for the immediately preceding regex
  - `{n}` means exactly *n* occurrences
  - `{n,}` means at least *n* occurrences
  - `{n,m}` means at least *n* occurrences but no more than *m* occurrences

- Example:
  - `.{0,}` same as `.*`
  - `a{2,}` same as `aaa*`

- If you want to group part of an expression so that `*` applies to more than just the previous character, use `( )` notation

- Subexpresssions are treated like a single character
  - `a*` matches `0` or more occurrences of `a`
  - `abc*` matches `ab`, `abc`, `abcc`, `abccc`, ...
  - `(abc)*` matches `abc`, `abcabc`, `abcabcabc`, ...
  - `(abc){2,3}` matches `abcabc` or `abcabcabc`

Regular Expressions: Some Practical Examples

- Variable names in C
  - `[a-zA-Z_][a-zA-Z_0-9]*`

- Dollar amount with optional cents
  - `\$[0-9]+(\.[0-9][0-9])?`

- Time of day
  - `(1[012]|[1-9]):[0-5][0-9] (am|pm)`

- HTML headers `<h1>` `<H1>` `<h2>` ...
  - `<[hH][1-4]>`

Regex Metacharacters

\b Matches a word boundary, that is, the position between a word and a space. For example, er\b matches the er in "never" but not the er in verb.
\B Matches a nonword boundary. ea*r\B matches the ear in never early.
\d Matches a digit character. Equivalent to [0-9].
\D Matches a nondigit character. Equivalent to [^0-9].
\f Matches a form-feed character.
\n Matches a newline character.
\r Matches a carriage return character.
\s Matches any white space including space, tab, form-feed, etc. Equivalent to [\f\n\r\t\v].
\S Matches any nonwhite space character. Equivalent to [^ \f\n\r\t\v].
\t Matches a tab character.
\v Matches a vertical tab character.
\w Matches any word character including underscore. Equivalent to [A-Za-z0-9_].
\W Matches any nonword character. Equivalent to [^A-Za-z0-9_].

```
 GNU nano 2.2.6                File: test01.sh

#!/bin/bash

echo -n "Your answer> "
read REPLY
if [[ $REPLY =~ ^[0-9]+$ ]]; then
    echo Numeric
else
    echo Non-numeric
fi
```

bash has a built-in regular expression comparison operator, represented by =~

تمرین: فایل زیر را به نحوی تغییر دهید که اینکه داده ی ورودی، آدرس IP است یا خیر را تشخیص دهد.

```
  GNU nano 2.2.6               File: IsIPAddress.sh

#!/bin/bash

if [ $# != 1 ]; then
    echo "Usage: $0 address"
    exit 1
else
    ip=$1
fi

if [[ $ip =~ ****************************** ]]; then
    echo "Looks like an IPv4 IP address"
else
    echo "oops"
fi
```

آشنایی با AWK

Programmable Text Processing with awk

- The awk utility scans one or more files and an action on all of the lines that match a particular condition.
- The actions and conditions are described by an awk program and range from the very simple to the complex.
- awk got its name from the combined first letters of its authors' surnames: Aho, Weinberger, and Kernighan.

Aho Weinberger Kernighan

- It borrows its control structures and expression syntax from the language C.

awk Program

- An awk program is a list of one or more commands of the form:

```
[ pattern ] [ \{ action \} ]
```

- For example:

```
BEGIN { print "List of html files:" }
/\.html$/ { print }                    ---> "/" then "\." then "html" then "$"
END { print "There you go!" }
```

- action is performed on every line that matches pattern (or condition in other words).
- If pattern is not provided, action is performed on every line.
- If action is not provided, then all matching lines are simply sent to standard output.
- Since patterns and actions are optional, actions must be enclosed in braces to distinguish them from pattern.
- The statements in an awk program may be indented and formatted using spaces, tabs, and new lines.

awk: Patterns and Actions

- Search a set of files for patterns.
- Perform specified actions upon lines or fields that contain instances of patterns.
- Does not alter input files.
- Process one input line at a time

- Every program statement has to have a pattern or an action or both
- Default pattern is to match all lines
- Default action is to print current record
- Patterns are simply listed; actions are enclosed in {}
- awk scans a sequence of input lines, or records, one by one, searching for lines that match the pattern
  - meaning of match depends on the pattern

awk: Patterns

- Selector that determines whether action is to be executed pattern can be:

- the special token BEGIN or END
- extended regular expressions (enclosed with //)
- arithmetic relation operators
- string-valued expressions
- arbitrary combination of the above:

/CSUN/ matches if the string "CSUN" is in the record

x > 0 matches if the condition is true

/CSUN/ && (name == "UNIX Tools")

Special awk Patterns: BEGIN, END

- BEGIN and END provide a way to gain control before and after processing, for initialization and wrap-up.
- BEGIN: actions are performed before the first input line is read.
- END: actions are done after the last input line has been processed.

```
BEGIN { print "List of html files:" }
/\.html$/ { print }
END { print "There you go!" }
```

awk: Actions

- action is a list of one or more of the following kinds of C-like statements terminated by semicolons:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression; conditional; expression ) statement
break
continue
variable = expression
print [ list of expressions ] [>expression]
printf format [, list of expressions] [>expression]
next (skips the remaining patterns on the current line of input)
exit (skips the rest of the current line)
{ list of statements }
```

- action may include arithmetic and string expressions and assignments and multiple output streams.

awk: An Example

```
$ ls | awk '
  BEGIN { print "List of html files:" }
  /\.html$/ { print }
  END { print "There you go!" }
  '
List of html files:
index.html
as1.html
as2.html
There you go!
$ _
```

awk: Variables

- awk scripts can define and use variables

```
BEGIN { sum = 0 }
{ sum ++ }
END { print sum }
```

- Some variables are predefined:

- NR - Number of records processed
- NF - Number of fields in current record
- FILENAME - name of current input file
- FS - Field separator, space or TAB by default
- OFS - Output field separator, space by default
- ARGC/ARGV - Argument Count, Argument Value array
  - Used to get arguments from the command line

awk: Fields

- Each input line is split into fields.
- Special variable FS: field separator: default is whitespace (1 or more spaces or tabs)

awk –Fc
  – sets FS to the character c
  – can also be changed in BEGIN

- $0 is the entire line
- $1 is the first field, $2 is the second field, ..., $NF is the last field
- Only fields begin with $, variables are unadorned

awk: Simple Output From AWK

- Printing Every Line
  – If an action has no pattern, the action is performed to all input lines

{ print }
  will print all input lines to standard out

{ print $0 }
  will do the same thing

- Printing Certain Fields
  – multiple items can be printed on the same output line with a single print statement

{ print $1, $3 }
  – expressions separated by a comma are, by default, separated by a single space when output

awk: Output (continued)

- Special variable NF: number of fields
  - Any valid expression can be used after a $ to indicate the contents of a particular field
  - One built-in expression is NF: number of fields

```
{ print NF, $1, $NF }
```

  - will print the number of fields, the first field, and the last field in the current record

```
{ print $(NF-2) }
```

  - prints the third to last field

- Computing and Printing
  - You can also do computations on the field values and include the results in your output

```
{ print $1, $2 * $3 }
```

awk: Output (continued)

- Printing Line Numbers
  - The built-in variable NR can be used to print line numbers

```
{ print NR, $0 }
```

  - will print each line prefixed with its line number

- Putting Text in the Output
  - you can also add other text to the output besides what is in the current record

```
{ print "total pay for", $1, "is", $2 * $3 }
```

  - Note that the inserted text needs to be surrounded by double quotes

```
pi@raspberrypi ~/session05/awk $ echo "hello world" | awk '
BEGIN {print "first word(total words):" }
END {print $1,NF}
'
first word(total words):
hello 2
pi@raspberrypi ~/session05/awk $ echo "hello world" | awk '
BEGIN {print "second word(total words):" }
END {print $2,NF}
'
second word(total words):
world 2
pi@raspberrypi ~/session05/awk $ _
```

awk: Fancier Output

- Lining Up Fields
  - like C, Awk has a printf function for producing formatted output
  - printf has the form:

printf(format, val1, val2, val3, ...)

{ printf("total pay for %s is $%.2f\n", $1, $2 * $3) }

  - when using printf, formatting is under your control so no automatic spaces or newlines are provided by awk. You have to insert them yourself.

{ printf("%-8s %6.2f\n", $1, $2 * $3) }

awk: Selection

- Awk patterns are good for selecting specific lines from the input for further processing

- Selection by Comparison

```
$2 >= 5 { print }
```

- Selection by Computation

```
$2 * $3 > 50 { printf("%6.2f for %s\n", $2 * $3, $1) }
```

- Selection by Text Content

```
$1 == "CSUN"
/CSUN/
```

- Combinations of Patterns

```
$2 >= 4 || $3 >= 20
```

- Selection by Line Number

```
NR >= 10 && NR <= 20
```

awk: Arithmetic and Variables

- awk variables take on numeric (floating point) or string values according to context.
- User-defined variables are unadorned (they need not be declared).
- By default, user-defined variables are initialized to the null string which has numerical value 0.

- awk Operators:

=	assignment operator; sets a variable equal to a value or string
==	equality operator; returns TRUE is both sides are equal
!=	inverse equality operator
&&	logical AND
||	logical OR
!	logical NOT
<, >, <=, >=	relational operators
+, -, /, *, %, ^	arithmetic
String concatenation	

awk: String Manipulation

- String Concatenation
  - new strings can be created by combining old ones

```
{ names = names $1 " " }
END { print names }
```

- Printing the Last Input Line
  - although NR retains its value after the last input line has been read, $0 does not

```
{ last = $0 }
END { print last }
```

awk: Built-In Functions

- awk contains a number of built-in functions.
- Arithmetic
  - sin, cos, atan, exp, int, log, rand, sqrt
- String
  - length, substitution, find substrings, split strings
- Output
  - print, printf, print and printf to file
- Special
  - system - executes a Unix command
    - e.g., system("clear") to clear the screen
    - Note double quotes around the Unix command
  - exit - stop reading input and go immediately to the END pattern-action pair if it exists, ot herwise exit the script

awk: Built-in Functions

- Example:

- Counting lines, words, and characters using length (a poor man's wc):

```
{
  nc = nc + length($0) + 1
  nw = nw + NF
}
END { print NR, "lines,", nw, "words,", nc, "characters" }
```

- substr(s, m, n) produces the substring of s that begins at position m and is at most n characters long.

awk: Control Flow Statements

- awk provides several control flow statements for making decisions and writing loops

- if-then-else

```
$2 > 6 { n = n + 1; pay = pay + $2 * $3 }
END {
  if (n > 0)
        print n, "employees, total pay is", pay, "average pay is", pay/n
  else
        print "no employees are paid more than $6/hour"
}
```

awk: Loops

- while

```
# interest1 - compute compound interest
# input: amount, rate, years
# output: compound value at end of each year
{ i = 1
  while (i <= $3)
  {
  printf("\t%.2f\n", $1 * (1 + $2) ^ i)
  i = i + 1
  }
}
```

- do-while

```
do {
  statement1
  } while (expression)
```

- for

```
# interest2 - compute compound interest
# input: amount, rate, years
# output: compound value at end of each year

{ for (i = 1; i <= $3; i = i + 1)
              printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

awk: Arrays

- Array elements are not declared
- Array subscripts can have any value:
  - numbers
  - strings! (associative arrays)

```
arr[3]="value"
grade["Korn"]=40.3
```

- Example

```
# reverse - print input in reverse order by line
{ line[NR] = $0 }    # remember each line
END {
 for (i=NR; (i > 0); i=i-1)
          { print line[i] }
}
```

awk: Examples

- We run a program that displayed the first, third, and last fields of every line:

```
$ cat awk2                          --> look at the awk script.
BEGIN { print "Start of file:", FILENAME }
{ print $1 $3 $NF }                 --> print first, third and last fields.
END { print "End of file" }
$ awk -f awk2 float                 --> execute the script.
Start of file: float
Wishwassky,
Myisstrong,
Andoftendays
Whenseemdedclear.
Nowwonderall...
End of file
$ _
```

awk: Examples

- In the next example, we display all of the lines that contained a t followed by an e, with any number of characters in between.

```
$ cat awk6                          --> look at the script.
/t.*e/  { print $0 }
$ awk -f awk6 float                 --> execute the script.
Wish I was floating in blue across the sky,
And I often visit the days
When everything seemed so clear.
Now I wonder what I'm doing here at all...
$ _
```

تمرین:

فایل awk01 را به نحوی اصلاح کنید که با اجرای دستور زیر، آدرس IP دستگاه را به همراه net mask نمایش داده شود.

```
pi@raspberrypi ~/session05/awk $ cat /etc/network/interfaces | awk -f awk01
```

```
  GNU nano 2.2.6                File: awk01

BEGIN{
print "start"
}
{
print $0
}
END{}
```

آشنایی با Sed

Transforming Files: sed

- Stream-oriented, non-interactive, text editor
- Look for patterns one line at a time and change lines accordingly
  - like awk.
- Non-interactive text editor
  - editing commands come in as script
  - there is an interactive editor ed which accepts the same commands
- A Unix filter
  - superset of previously mentioned tools

sed vs. awk

- sed is a pattern-action language, like awk
- awk processes fields while sed only processes lines

- sed +
  - regular expressions
  - fast
  - concise
- sed –
  - hard to remember text from one line to another
  - not possible to go backward in the file
  - no way to do forward references like /..../+1
  - no facilities to manipulate numbers
  - cumbersome syntax
- awk +
  - convenient numeric processing
  - variables and control flow in the actions
  - convenient way of accessing fields within lines
  - flexible printing
  - C-like syntax

Sed Usage

- Edit files too large for interactive editing
- Edit any size files where editing sequence is too complicated to type in interactive mode
- Perform "multiple global" editing functions efficiently in one pass through the input
- Edit multiples files automatically
- Good tool for writing conversion programs

Conceptual overview

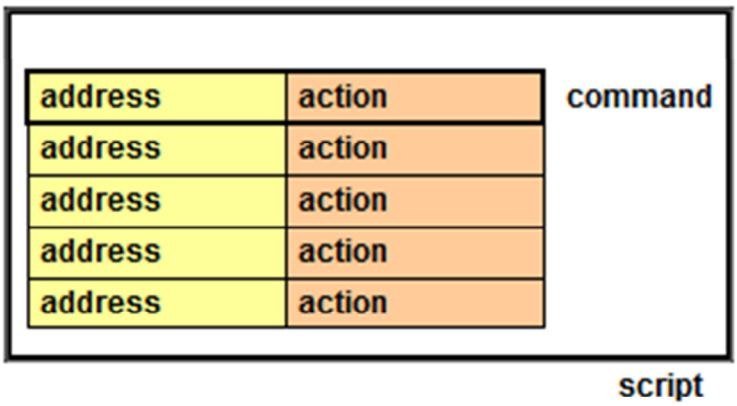
- A script is read which contains a list of editing commands
  - Can be specified in a file or as an argument
- Before any editing is done, all editing commands are compiled into a form to be more efficient during the execution phase.
- All editing commands in a sed script are applied in order to each input line.
- If a command changes the input, subsequent command address will be applied to the current (modified) line in the pattern space, not the original input line.
- The original input file is unchanged (sed is a filter), and the results are sent to standard output (but can be redirected to a file).

sed Syntax

- sed [-n] [-e] ['command'] [file...]
- sed [-n] [-f scriptfile] [file...]
- -n - only print lines specified with the print command (or the 'p' flag of the substitute ('s') command)
- -f scriptfile - next argument is a filename containing editing commands
- -e command - the next argument is an editing command rather than a filename, useful if multiple commands are specified
- If the first line of a scriptfile is "#n", sed acts as though -n had been specified

Scripts

- A script is nothing more than a file of commands
- Each command consists of an address and an action, where the address can be a regular expression or line number.

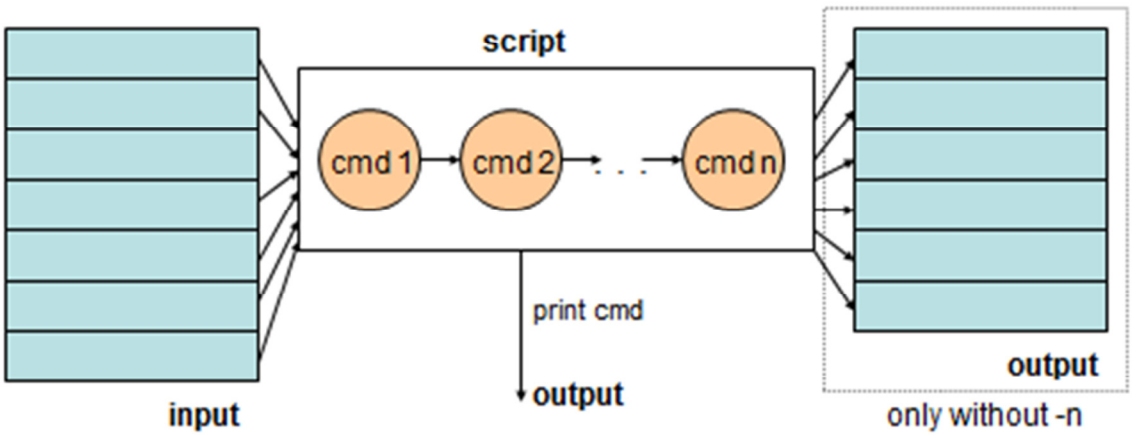


Scripts (continued)

- As each line of the input file is read, sed reads the first command of the script and checks the address against the current input line:
  - if there is a match, the command is executed
  - if there is no match, the command is ignored
  - sed then repeats this action for every command in the script file
- When it has reached the end of the script, sed outputs the current line (pattern space) unless the -n option has been set

Sed Flow of Control

- sed then reads the next line in the input file and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input file



sed Commands

- sed commands have the general form

[address[, address]][!]command [arguments]

- sed copies each input line into a pattern space
  - if the address of the command matches the line in the pattern space, the command is applied to that line
  - if the command has no address, it is applied to each line as it enters pattern space
  - if a command changes the line in pattern space, subsequent commands operate on the modified line
- When all commands have been read, the line in pattern space is written to standard output and a new line is read into pattern space

Addressing

- An address can be either a line number or a pattern, enclosed in slashes (/pattern/)
- A pattern is described using regular expressions (BREs, as in grep)
- If no pattern is specified, the command will be applied to all lines of the input file
- To refer to the last line: $
- Most commands will accept two addresses
  - If only one address is given, the command operates only on that line
  - If two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively
- The ! operator can be used to negate an address; i.e., address!command causes command to be applied to all lines that do not match address

Commands

- command is a single letter
- Example:

Deletion: d

[address1][,address2]d

- Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
- A new line of input is read and editing resumes with the first command of the script.

Address and Command Examples: delete

d	deletes all lines
6d	deletes line 6
/^$/d	deletes all blank lines
1,10d	deletes lines 1 through 10
1,/^$/d	deletes from line 1 through the first blank line
/^$/,$d	deletes from the first blank line through the last line of the file
/^$/,10d	deletes from the first blank line through line 10
/^ya*y/,/[0-9]$/d	deletes from the first line that begins with yay, yaay, yaaay, etc through the first line that ends with a digit

Multiple Commands

- Braces {} can be used to apply multiple commands to an address

```
[address][,address]{
  command1
  command2
  command3
}
```

- The opening brace must be the last character on a line
- The closing brace must be on a line by itself
- Make sure there are no spaces following the braces

- Alternatively, use ";" after each command:

```
[address][,address]{command1; command2; command3; }
```

- Or:

```
'[address][,address]command1; command2; command3'
```

Sed Commands

- Although sed contains many editing commands, we are only going to cover the following subset:

s - substitute
a - append
i - insert
c - change
d – delete
p - print
r - read
w - write
y - transform
= - display line number
N - append the next line to the current one
q - quit

Print

- The print command (p) can be used to force the pattern space to be output, useful if the -n option has been specified
- Syntax:

[address1[,address2]]p

- Note: if the -n or #n option has not been specified, p will cause the line to be output twice!
- Examples:

1,5p	will display lines 1 through 5
/^$/,$p	will display the lines from the first blank line through the last line of the file

Substitute

- Syntax:

[address(es)]s/pattern/replacement/[flags]

- pattern - search pattern
- replacement - replacement string for pattern
- flags - optionally any of the following

n	a number from 1 to 512 indicating which occurrence of pattern should be replaced
g	global, replace all occurrences of pattern in pattern space
p	print contents of pattern space

Substitute Examples

s/Puff Daddy/P. Diddy/

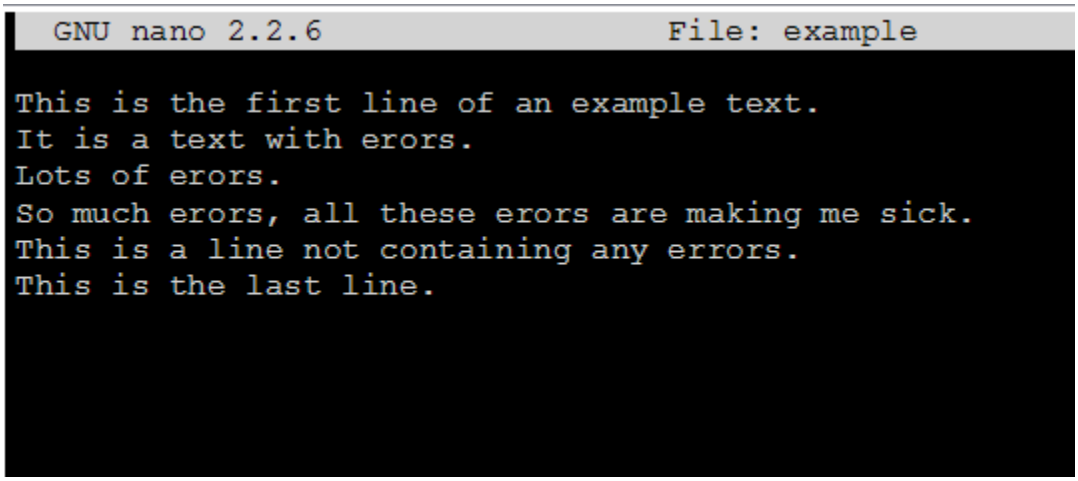
Substitute P. Diddy for the first occurrence of Puff Daddy in pattern space

s/Tom/Dick/2

Substitutes Dick for the second occurrence of Tom in the pattern space

s/wood/plastic/p

Substitutes plastic for the first occurrence of wood and outputs (prints) pattern space



```
pi@raspberrypi ~/session05/sed $ sed -n '/erors/p' example
It is a text with erors.
Lots of erors.
So much erors, all these erors are making me sick.
pi@raspberrypi ~/session05/sed $ _
```

Sed options

Option	Effect
`-e SCRIPT`	Add the commands in SCRIPT to the set of commands to be run while processing the input.
`-f`	Add the commands contained in the file SCRIPT-FILE to the set of commands to be run while processing the input.
`-n`	Silent mode.
`-V`	Print version information and exit.

```
pi@raspberrypi ~/session05/sed $ sed -e 's/erors/errors/g' -e 's/last/final/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.
pi@raspberrypi ~/session05/sed $
```

```
pi@raspberrypi ~/session05/sed $ sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with erors.
> Lots of erors.
> So much erors, all these erors are making me sick.
> This is a line not containing any errors.
> This is the last line.
pi@raspberrypi ~/session05/sed $
```

Replacement Patterns

- Substitute can use several special characters in the replacement string

&	replaced by the entire string matched in the regular expression for pattern
\n	replaced by the nth substring (or subexpression) previously specified using "\(" and "\)"
\	used to escape the ampersand (&) and the backslash (\)

Replacement Pattern Examples

"the UNIX operating system ..."

```
s/.NI./wonderful &/
```

--> "the wonderful UNIX operating system ..."

```
$ cat test1
first:second
one:two
$ sed 's/\(.*\):\(.*\)/\2:\1/' test1
second:first
two:one
```

"unix is fun"

```
sed 's/\([[:alpha:]]\)\([^ \n]*\)/\2\1ay/g'
```

--> "nixuay siay unfay"

Append, Insert, and Change

- Syntax for these commands is a little strange because they must be specified on multiple lines

- append

```
[address]a\
text
```

- insert

```
[address]i\
text
```

- change

```
[address(es)]c\
text
```

- append/insert for single lines only, not range

Append and Insert

- Append places text after the current line in pattern space

- Insert places text before the current line in pattern space

  – each of these commands requires a \ following it
  – text must begin on the next line.
  – if text begins with whitespace, sed will discard it unless you start the line with a \

- Example:

```
/<Insert Text Here>/i\
Line 1 of inserted text\
    <-- whitespace -> Line discarded by sed\
\       Line 2 of inserted text
```

would leave the following in the pattern space:

```
Line 1 of inserted text
      Line 2 of inserted text
<Insert Text Here>
```

Change

- Unlike Insert and Append, Change can be applied to either a single line address or a range of addresses
- When applied to a range, the entire range is replaced by text specified with change, not each line
- Exception! If the Change command is executed with other commands enclosed in { } that act on a range of lines, each line will be replaced with text
- No subsequent editing allowed

Change Examples

- Remove mail headers from email messages
  - the address specifies a range of lines beginning with a line that begins with From until t he first blank line.
- The first example replaces all lines with a single occurrence of <Mail Header Removed>.

```
/^From /,/^$/c\
<Mail Headers Removed>
```

- The following replaces each line with <Mail Header Removed>

```
/^From /,/^$/{
        s/^From //p
        c\
        <Mail Header Removed>
}
```

Using !

- If an address is followed by an exclamation point (!), the associated command is applied to all lines that don't match the address or address range

- Examples:

1,5!d delete all lines except 1 through 5

/black/!s/cow/horse/
substitute "horse" for "cow" on all lines
except those that contained "black"

- e.g.

"The brown cow" -> "The brown horse"
"The black cow" -> "The black cow"

Reading and Writing from/to Files

```
$ cat tmp
one two three
one three five
two four six
$ sed 'r tmp'
My first line of input          ---> no read until the first line is taken from the input
My first line of input
one two three
one three five
two four six
My next line
My next line^D
$ sed 'w tmp1'
hello 1
hello 1
hello 2
hello 2
hello 3
hello 3^D
$ cat tmp1
hello 1
hello 2
hello 3
$ _
```

Playing with Line numbers

```
$ sed '=' tmp
1
hello 1
2
hello 2
3
hello 3
$ sed -n '=' tmp1
1
2
3
$ _
```

Concatenating a Line with the Next Line

- Concatenating subsequent lines is done with the N command that appends the following line into the pattern space together with the separating new line character.

```
$ cat tmp1
hello 1
hello 2
hello 3
$ sed '=' tmp1 | sed '{N; s/\n/<NL>/g;}'
1<NL>hello 1
2<NL>hello 2
3<NL>hello 3
$ _
```

Transform

- The Transform command (y) operates like tr, it does a one-to-one or character-to-character replacement
- Transform accepts zero, one or two addresses

[address[,address]]y/abc/xyz/

- every a within the specified address(es) is transformed to an x, b to y and c to z

y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/

changes all lower case characters on
the addressed line to upper case

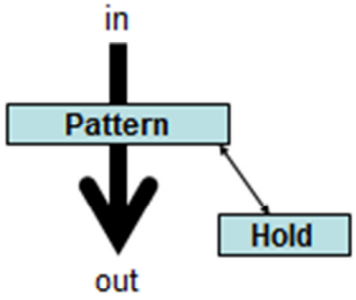
- if you only want to transform specific characters (or a word) in the line, it is much more difficult and requires use of the hold space

Pattern and Hold spaces

- Pattern space: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- Hold space: Secondary temporary buffer for temporary storage only

h – put pattern space into the hold space
H – append

g – get the hold space into the pattern space
G - append



x – exchange the contents of the hold space and the pattern space

Hold Buffer Example

- Edit a line, print the change and show the original

```
$ sed '/Unix/{h; s/.* Unix \(.*\) .*/\1:/; p; x; }'
This describes the Unix ls command.
ls:
This describes the Unix ls command.
This describes the Unix cp command.
cp:
This describes the Unix cp command.
$ sed '/Unix/{G;}'
This describes the Unix ls command.
This describes the Unix ls command.
                                    ---> G takes empty line from the hold buffer.
Something else.
Something else.
Again.
Again.
```

Quit

- Quit causes sed to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This can be used to save time when you only want to process some portion of the beginning of a file
- Example
- to print the first 100 lines of a file (like head)

```
sed '100q' filename
```

- sed will, by default, send the first 100 lines of filename to standard output and then quit processing